Introduction

Useful information

- My name: Giorgio Metta
- My email: pasa@liralab.it
- Office/Lab: 010 71781 411
- Where: LIRA-Lab, Villa Bonino, Ground Floor (su appuntamento)
- Web site: http://www.liralab.it/os
- Mailing list: os@liralab.it

Outline of the course

- Processes, threads, scheduling
- IPC
- Memory management
- I/O
- Filesystem
- Embedded systems

- The exam consists of:
  - 1 problem set
    - C++ programming 1/3
  - 1 oral exam:
    - Theory and short exercises 2/3

Background

- Required
  - Programming C/C++
- Helpful
  - Linux/Unix, Windows
- Main idea is to learn, so, don't freak out even if it might seem hard!

References

- Andrew S. Tanenbaum, *Modern operating systems,* Prentice Hall International 2001. ISBN: 0-13-092641-8

This slide is intentionally left blank

Operating system

- What's inside the computer?
  - Layers:

Web browser	Banking system	Airline reservation	} application programs
Compilers	Editors	Command interpreter (shell)	} system programs
Operating system			
Machine language			} hardware
Microarchitecture			
Physical devices			

Meaning of the layers

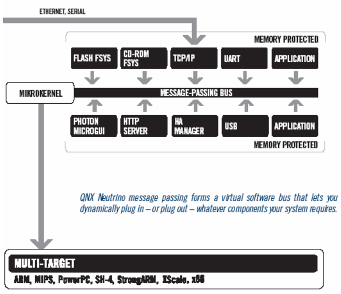
- Physical devices: self explaining
- Microarchitecture: define data path within the microprocessor (using registers) sometimes using a microprogram
- Machine language/Assembly language: instruction set (e.g. 50-300 instructions)

Where does the OS start?

Kernel mode Supervisor mode	User mode
Hardware protection (on modern microprocessors)	
All instructions allowed	Certain instructions not allowed
Timer interrupt handler	Compiler, editor, web browser

Example: microkernel OS

Microkernel RTOS. In QNX Neutrino, only the most fundamental OS primitives (e.g. signals, timers, scheduling) are handled in the kernel itself. All other components – drivers, file systems, protocol stacks, user applications – run outside the kernel as separate, memory-protected processes.



Operating system's job

- Manage the hardware (all the devices)
- Provide user programs with simpler interface to the hardware (extended machine)

Example: floppy drive

- Specific chip (NEC PD765)
- 16 different commands
- Load between 1 and 9 bytes into a device register
- Read/Write require 13 parameters packed into 9 bytes
- Reply from the device consists of 7 bytes (23 parameters)
- Control of the motor (on/off)

Abstraction

- Better to think in terms of *files* with *names* rather than specific floppy drive commands
- Other unpleasant hardware:
  - Interrupts
  - Timers
  - Memory management
  - …
- **Extended or virtual machine**

OS as resource manager

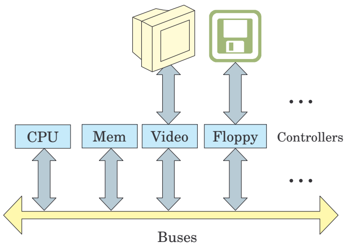
- Allocation of resources:
  - Processors, memory, I/O devices among a set of programs competing for them
- Example: allocating the printer
  - Buffering output rather than just print at random
- Multiple users: sharing of resources and avoid conflicts (share vs. security)

Sharing

- *Time* and *space* multiplexing
- Multiplexing in time: e.g. printer, processor
  - Print one job at a time
- Multiplexing in space: e.g. memory, disks
  - Divide memory among many processes

Computer hardware

- Processors
- Memory
- I/O devices
- Buses



Processors

Registers
- Program counter (PC): next instruction
- Stack pointer (SP): stack in memory

} Context switch

- Program Status Word (PSW): condition bits (e.g. kernel vs. user mode)
- Base register: relocation of executables

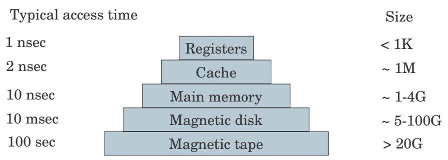
System call
- SW interrupt
- From User to Kernel mode

Complexity of the CPU HW
- Pipeline architecture
- Superscalar

Memory

- Ideally…
  - Extremely fast (faster than the CPU in executing an instruction)
  - Abundantly large
  - Dirt cheap

Memory (for real)



Memory cntd.

- Registers: typical 32 in a 32 bit CPU
- Cache: divided into cache lines (64 bytes each)
  - Cache hit – no main memory access, no bus involvement
  - Cache miss – costly
- Main memory
- Disk (multiple plates, heads, arms)
  - Logical structure: sectors, tracks, cylinders
- Magnetic tape: backup, cheap, removable

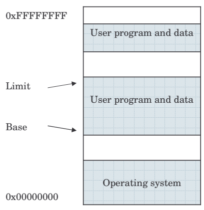
Multiple programs in memory

- Base and Limit register
- Hardware support for relocation and multiple programs in memory

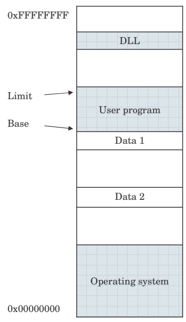
Fetch:

Instruction
if (PC<Limit) Fetch(PC+Base)
else Troubles(SigFault)

Data
if (Addr<Limit) Fetch(Addr+Base)
else Troubles(SigFault)



DLL's (in principle)



- Requires an MMU with multiple Base/Limit register pairs

Memory Management Unit

- Managing the MMU is one of the OS tasks:
  - Balancing context switches since they impact on performances: e.g. MMU registers have to be saved, cache emptied, etc.

I/O devices

- Usually a controller + the actual device
  - For example: a disk controller may hide the details of driving the arm and heads to the appropriate location to read a certain piece of data
  - Sometimes the controller is a small embedded microprocessor in itself
- The interface to the OS is somewhat standardized:
  - IDE disk drives conform to a standard
- Device driver: a piece of the OS. Device drivers run in kernel mode since they have to access I/O instructions and device registers

Device drivers

1. Unix. Compiled and linked with the kernel (although Linux supports dynamic loading of DD)
2. Windows. An entry into an OS table. Loaded at boot
3. Dynamic. USB, IEEE1394 (firewire). At boot time the OS detects the hardware, finds the DD, and loads them

I/O registers

- E.g. small number of registers used to communicate
- Memory mapped: the registers appear at particular locations within the OS address space
- I/O instructions: some CPUs have special privileged (kernel mode) I/O instructions (IN/OUT). Registers are mapped to special locations in I/O space

Ways of doing I/O

1. Polling
2. Interrupt based
3. DMA

Polling

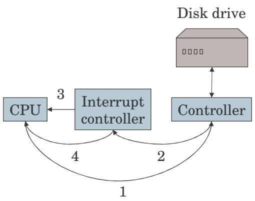
- User makes a system call
- OS calls DD
- DD talks to device, prepares I/O, starts I/O and sits waiting (busy waiting) for I/O completion

- *Busy waiting* means that the CPU is busy polling a flag

Interrupt

- A piece of hardware called "interrupt controller"

1. CPU issues the I/O request via the device driver
2. On termination the device signals the CPU's interrupt controller (if the interrupt controller is not busy servicing another higher priority interrupt)
3. If the interrupt can be handled then the controller asserts a pin on the CPU.
4. The interrupt controller puts the address of the device into the bus



Interrupt (cntd.)

- When the CPU decides to take the interrupt:
  - Stores registers (push them into the stack)
  - Switches into kernel mode
  - Uses the device's address to index a table (interrupt vector)
  - Calls the handler contained at the location located in the interrupt vector
  - Once the handler is executed it returns from the handler by popping the registers from the stack

Direct Memory Access DMA

- Yet another piece of hardware: DMA controller
  - Communication between memory and device can be carried out by the DMA controller with little CPU intervention
  - When the DMA is completed the controller asserts an interrupt as before

Buses

- Multiple buses (cache, local, memory, PCI, USB, IDE…)
- OS must be aware of all of them to manage things appropriately
- Plug&Play – dynamic allocation of I/O and memory addresses (BIOS code)

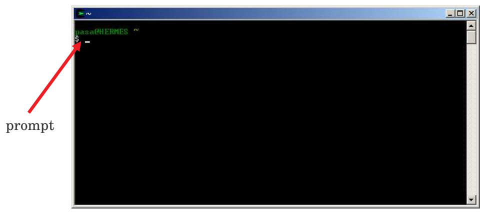
This slide is intentionally left blank

Concepts

- Processes
- Deadlocks
- Memory management
- I/O
- Files
- Security
- …

The Shell

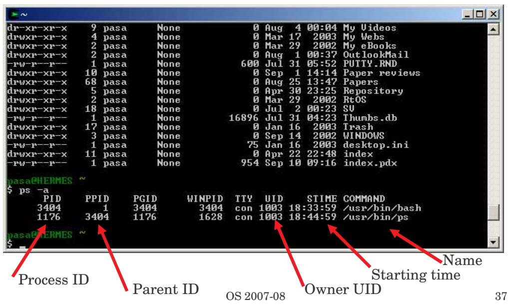
- Unix command interpreter (or similarly the "command" in windows)
- Clearly, it's not part of the OS



Processes

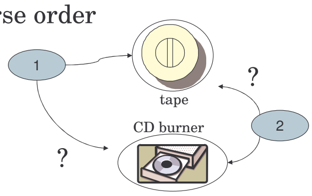
- Associated with each process:
  - Address space (program + data + stack)
  - Entry into the process table (a list of processes)
    - Set of registers (e.g. PC, PSW, etc.)
    - MMU status, registers
- Processes can be created, terminated, signaled (SW interrupt)
- They form a tree (a hierarchy) on some systems
- Process cooperation is obtained by means of IPC (inter-process communication) mechanisms
- Processes start with the privileges of the user who starts them

ps (process status) command



Deadlocks

- Two or more processes mutually requesting the same set of resources
- Example: two processes trying to use simultaneously a tape and CD burner in reverse order



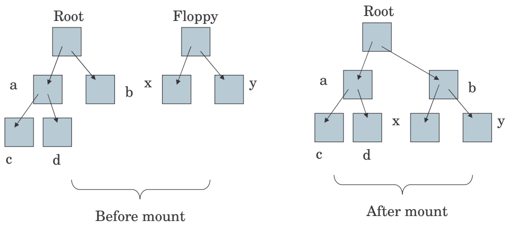
Memory management

- Virtual memory
  - Allowing processes requesting more memory than the computer main memory to run
  - Swap space/swapping. Storing some of the process' memory in the disk

Files

- Concept of directory (group files together)
- A tree-like structure similar to the process hierarchy
- A file is specified by its *path* name
  - E.g. /usr/bin/ps
- In UNIX there's a *root* directory (/)
  - Windows has a root for each drive: A:, B:, C:, etc.
- Working directory (a process property)
  - Where path not beginning with slash are looked for
- Interface between OS and program code is through a small integer called *file descriptor*

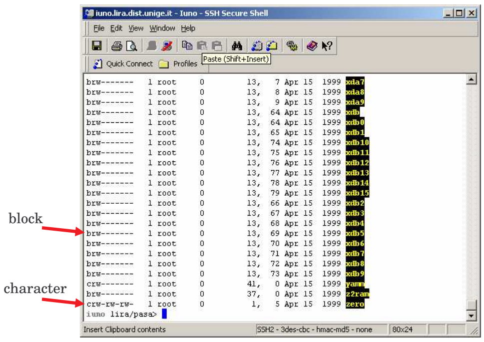
mount



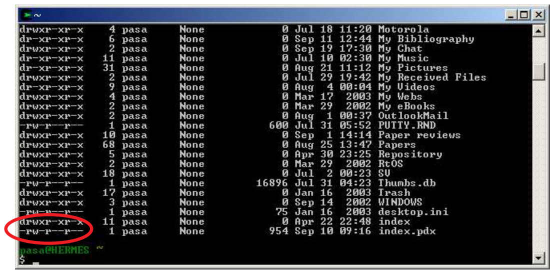
Special file

- A device driver gets a special entry into the file system (usually under /dev)
- Block special files
  - Randomly addressable blocks: a disk
- Character special files
  - A stream of character data: modem, printer

Special file (ctnd.)



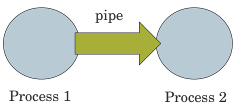
Security



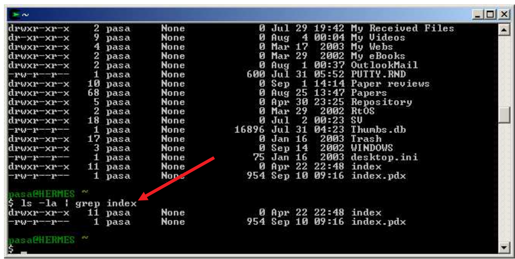
-rwxrwxrwx

Pipe

- It's a sort of pseudofile
- Allows connecting two processes as they were issuing read/write system calls to a regular file



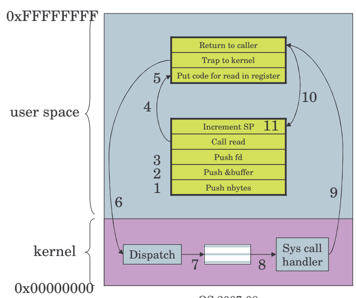
Pipe example



This slide is intentionally left empty

System calls

```
count = read(fd, buffer, nbytes);
```



System calls

```
count = read(fd, buffer, nbytes);
```

1. Push nbytes into the stack
2. Push buffer into the stack
3. Push fd into the stack
4. Library calls `read`
5. Put sys call code into register
6. Trap to kernel
7. Examines the call code, query table
8. Call handler, execute `read` code
9. Return to caller (maybe)
10. Pop stack (i.e. increment SP)
11. Continue execution

read(2) - Linux man page

NAME

read - read from a file descriptor

SYNOPSIS

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, **read()** returns zero and has no other results. If *count* is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read()** was interrupted by a signal. On error, -1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

ERRORS

EINTR
The call was interrupted by a signal before any data was read.
EAGAIN
Non-blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.
EIO I/O error. This will happen for example when the process is in a background process group, tries to read from

EAGAIN
Non-blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.
EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling tty, and either it is ignoring or blocking SIGTTIN or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape.
EISDIR
fd refers to a directory.
EBADF
fd is not a valid file descriptor or is not open for reading.
EINVAL
fd is attached to an object which is unsuitable for reading.
EFAULT
buf is outside your accessible address space.

Other errors may occur, depending on the object connected to *fd*. POSIX allows a **read** that is interrupted after reading some data to return -1 (with *errno* set to EINTR) or to return the number of bytes already read.

CONFORMING TO

SVr4, SVID, AT&T, POSIX, X/OPEN, BSD 4.3

RESTRICTIONS

On NFS file systems, reading small amounts of data will only update the time stamp the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave atime updates to the server and client side reads satisfied from the client's cache will not cause atime updates on the server as there are no server side reads. UNIX semantics can be obtained by disabling client side attribute caching, but in most situations this will substantially increase server load and decrease performance.

Many filesystems and disks were considered to be fast enough that the implementation of **O_NONBLOCK** was deemed unneccesary. So, O_NONBLOCK may not be available on files and/or disks.

SEE ALSO

close(2), **fcntl**(2), **ioctl**(2), **lseek**(2), **readdir**(2), **readlink**(2), **select**(2), **write**(2), **fread**(3), **readv**(3)

System call interface (part of)

Call	Description
`pid = fork()`	Create a child process identical to the parent
`pid = waitpid(pid, &statloc, options)`	Wait for a child to terminate
`s = execve(name, argv, environp)`	Replace a process' core image
`exit(status)`	Terminate process execution and return status
`fd = fopen(file, how, ...)`	Open a file for reading, writing or both
`s = close(fd)`	Close an open file
`n = read(fd, buffer, nbytes)`	Read data from a file into a buffer
`n = write(fd, buffer, nbytes)`	Write data from a buffer into a file
`position = lseek(fd, offset, whence)`	Move the file pointer
`s = stat(name, &buf)`	Get a file's status information
`s = mkdir(name, mode)`	Create a new directory
`s = rmdir(name)`	Remove an empty directory
`s = link(name1, name2)`	Create a new entry, name2 pointing to name1
`s = ulink(name)`	Remove a directory entry
`s = mount(special, name, flag)`	Mount a file system
`s = umount(special)`	Unmount a file system

System call interface (cntd.)

Call	Description
`s = chdir(dirname)`	Change the working directory
`s = chmod(name, mode)`	Change a file's protection bits
`s = kill(pid, signal)`	Send a signal to a process
`seconds = time(&seconds)`	Get elapsed time in seconds since Jan 1st, 1970

Process management

```
while (1)
{
        type_prompt();
        read_command(command, parameters);

        if (fork() != 0)
        {
                waitpid(-1, &status, 0);
        }
        else
        {
                execve(command, parameters, 0);
        }
}
```

lseek

```
position = lseek(fd, offset, whence)
```

- Random access to a file
- Imagine the file as accessed through a pointer
- `lseek` moves the pointer

Directory (in UNIX)

- Each file is identified by an *i-number*
- The *i-number* is an index into a table of *i-nodes*
- A directory is a file containing a list of *i-number – ASCII name*

Link

- Called a shortcut in some versions of Windows

/usr/ast	
16	mail
81	games
40	test

/usr/jim	
31	bin
70	memo
38	prog1

```
link("/usr/jim/memo", "usr/ast/note")
```

/usr/ast	
16	mail
81	games
40	test
70	note

/usr/jim	
31	bin
70	memo
38	prog1

Win32 API

- Different philosophy
- Many calls (API – Application Program Interface), not all of them are actually system calls
- GUI included into the API (in comparison X-Windows is all user level code)

Example of Win32

fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	None	CreateProcess does the job
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributeEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	None	
unlink	DeleteFile	Destroy an existing file
mount	None	
umount	None	
chdir	SetCurrentDirectory	Change the current working directory
chmod	None	
kill	None	
time	GetLocalTime	Get the current time

Operating system structure

- Monolithic systems
- Layered systems
- Virtual machines
- Exokernels
- Client-Server model

Monolithic systems

- The "big mess"
- No organized structure
- A bit of structure anyway:
  - System calls requires parameters in a well defined place (e.g. the stack)
  - Three layers:
    - Application program
    - Service procedures
    - Helper procedures

Layered systems

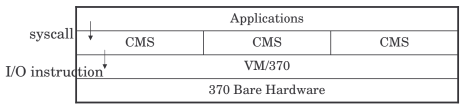
- Each layer relies only on services provided by lower level layers

Layer	Function
5	User/operator
4	User programs
3	I/O management
2	Operator-process communication
1	Memory and disk management
0	Processor allocation and multiprogramming

Virtual machines

- Timesharing provides:
  - Multiprogramming
  - Extended machine
- Decouple the two functions:
  - Virtual machine monitor (a SW layer)
  - It does the multiprogramming providing a "simulation" of the bare HW
- On top of the monitor any compatible OS could be run
- Also the Pentium (8086 mode, running DOS applications) and Java VM provide a similar mechanism (slightly different though)

Virtual machines



Exokernel

- Each process is given a subset of the resources (at any given moment) and NOT a simulation of the whole machine
- Simpler
- Saves a layer of mapping
- Each VM in this case is given a subset of memory, disk space, etc.
- The OS checks for conflicts

Client-Server model

- Microkernel
- Services are moved into user-space processes (e.g. the filesystem)
- The kernel handles message passing mechanisms to make communication possible between user code and services
- Easy to "remote" the message passing (distributed system)
- Resilient: a crash in one module doesn't compromise the whole system (which can then recover from the crash)
- I/O and HW access must be done into the kernel (spoils a bit the nice client-server model) for example in device drivers

Example: microkernel OS

Microkernel RTOS. In QNX Neutrino, only the most fundamental OS primitives (e.g. signals, timers, scheduling) are handled in the kernel itself. All other components – drivers, file systems, protocol stacks, user applications – run outside the kernel as separate, memory-protected processes.